Register Allocation

code generation until now

- When creating temporaries as you generate code, each temporary gets to sit in its own register
  - Temporaries are essentially “virtual” registers
- Real machines have a limited number of registers available for general purpose use
  - Called **architectural** registers — registers addressable by instructions (machines may have many more internal registers)
  - RISC-V: 32 integer registers and 32 floating point registers
    - But many of these registers are reserved for special purposes (stack pointer, frame pointer, return address, etc.)
    - In practice, fewer registers available
- What do you do with temporaries?

wouldn't it be nice

- One extreme: all temporaries are registers
  - No loads or stores required for temporaries
- All variables/local variables loaded in to registers at the beginning of a function, saved back to memory at the end of the function
  - No "extra" loads and stores required for multiple uses of the same variable
- But this runs into the limit on the number of registers!

simple code generation

- Code generation uses a lot of temporaries; treat each temporary as a local variable that gets a spot on the stack
- Generate code for operations on temporaries the same way you generate code for operations on variables:
  - Load temporaries into registers
  - Perform operation
  - Store result in temporary
- How many registers does this need?
  - Why is this bad?

middle ground

- One extreme doesn't work (cannot keep all values in registers)
- The other extreme isn't efficient (don't want to keep loading/storing values)

- What if we pick some temporaries and variables to keep in registers?
  - Use registers for values we need, or need often
  - If we run out of space in registers, can **spill** registers to the stack (essentially, go back to treating it as a local variable)

- This is **register allocation**

Global vs. local

- Same distinction as global vs. local register allocation
  - Local register allocation is for a single basic block (BB)
  - Global register allocation is for an entire function (but not interprocedural – why?)
- Will cover some local allocation strategies now, global allocation later

naïve register allocation

- For each basic block
  - Find the number of references of each variable
  - Assign registers to variables with the most references

- Details
  - Keep some registers free for operations on unassigned variables and spilling
  - Store **dirty** registers at the end of BB (i.e., registers which have variables assigned to them and whose value has changed)
  - Do not need to do this for temporaries (why?)

drawbacks

- Suppose we only have two "extra" registers for this code →
- What variables go into registers?
- Could we do better?

```
1: T1 = A + B
2: T2 = A + T1
3: T3 = A + T2
4: D = A + T3
5: T4 = C + B
6: T5 = T4 + D
7: E = T5 + D
```

drawbacks

- Suppose we only have two “extra” registers for this code →

- What variables go into registers?
- Could we do better?

- Variables/temporaries that are **dead** do not need to be in registers anymore!
  - A and D can share a register
  - And so can all the temporaries!

```
T1 = A + B
T2 = A + T1
T3 = A + T2
D = A + T3
T4 = C + B
T5 = T4 + D
E = T5 + D
```

next: local register allocation